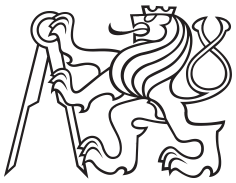
Bachelor's Thesis



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

Manuscripts Classification Using Convolutional Deep Learning Networks

Transfer learning

Zuzana Kožnarová

Supervisors: Dipl.-Ing. Vincent Christlein,
Prof. Dr.-Ing. Andreas Maier,
doc. Ing. Daniel Novák, Ph.D.
Field of study: Cybernetics and Robotics
Subfield: Robotics
August 2017

Acknowledgements

I would like to express my gratitude to my supervisor Dipl.-Ing. Vincent Christlein for relevant feedback to my research related to my bachelor thesis. I would like to thank people from BlindShell, specifically doc. Ing. Daniel Novák, Ph.D. and Ing. Jan Hadáček, for their help with my project last semester.
Many thanks to my family that helped me every time I needed, not only during last 3 years at the university. I want to thank my father, who was my example for studying at the Faculty of Electrical Engineering (FEL), also to my mother, who is embodiment of understanding and good, and to my aunt Jarmila, who taught me English.
Last but not least, I would like to thank my partner Jakub Havlíček and my friends, concretely David Sedláček, Martin Saslík, Václav Veselý, David Kopecký, Martin Vlašimský and Michal Stračinský, who were my comrades-in-FEL.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

.....

Abstract

The thesis concentrates on improving recognition performance for small datasets using convolutional neural networks (CNN). Specifically, it is about type classification of handwritings in Latin. The main topics are the use of pre-training and the influence of pre-training on augmented datasets. In the thesis, the impact of learning-rate changes and also the effect of fixing weights in several layers of the beginning of the network are examined. Last but not least, there is space devoted to preprocessing of the dataset using whitening. The whitening is used in the context of individual images.

The CNN models VGG16 [2] and Resnet50 [1] were used, as they were applied to Latin texts classification with one of the highest precision (architecture VGG16 reached accuracy of 0.7649).

The goal of these and other methods is to achieve improvement in classification against the results of the Classification of Medieval Handwritings in Latin 2016 Script competition [3].

Keywords: CNN, Convolutional Deep Learning Networks, Medieval handwritings in Latin script, pre-training, fine-tuning, CLAMM12, ICDAR16, supervised learning, dataset

Supervisors:
Dipl.-Ing. Vincent Christlein,
Prof. Dr.-Ing. Andreas Maier,
doc. Ing. Daniel Novák, Ph.D.

Abstrakt

Práce se zabývá možností zlepšení přesnosti klasifikace pro malé datové sady za použití konvolučních neuronových sítí (CNN). Konkrétně se jedná o klasifikaci typu historických ručně psaných latinských textů. Hlavními tématy je využití pretrainingu a vliv pretrainingu na rozšířené datové sady. V práci je dále sledován dopad změn learningratu a také účinek zafixování vah pro prvních několik vrstev. V neposlední řadě pak je věnován prostor preprocesingu datové sady pomocí whiteningu. Whitening je použit v kontextu jednotlivých obrázků.

Byly použity modely CNN VGG16 [1] a Resnet50 [2], jelikož byly již v minulosti aplikovány pro klasifikaci latinských textů s jednou z nejvyšších přesností (architektura VGG16 dosáhla přesnosti 0.7649).

Cílem těchto a dalších metod je dosáhnout zlepšení přesnosti klasifikace vůči výsledkům ze soutěže Classification of Medieval Handwritings in Latin 2016 Script [3].

Klíčová slova: CNN, konvoluční hluboké neuronové sítě, středověké ručně psané latické texty, předtrénování, dotrénování, CLAMM12, ICDAR16, učení s učitelem, datová sada

Překlad názvu: Aplikace hlubokých konvolučních neuronových sítí pro klasifikaci rukopisů — Transfer learning

Contents

Figures

Tables

Chapter 1

Introduction

In recent years, deep learning techniques based on neural networks (NNs) [4] have set new performance standards for a wide variety of machine learning or computer vision tasks. However, one problem of these deep NNs is that they need many training samples. Thus, for small or medium-sized datasets, classical computer vision approaches still surpass NNs. For example for the task of medieval script type classification [5], methods based on CNNs do not outperform classical approaches based on "bag of visual words" [3]. The goal of this work is to improve the performance of CNNs for script type classification. A possible solution to achieve better results is to pre-train CNNs for a different, yet similar task and fine-tune it later with training data of the original task.

In detail, the thesis consists of the following parts: implementation of a CNN system for script style classification, which is based on existing work by Mike Kestemont [6], evaluation of usefulness of pre-training for script-style classification using other datasets and investigation of other methods to improve recognition performance in tasks with scarce data.

The implementation was done in Python using the deep learning frameworks Keras [7] and Tensorflow [8].

1.1 Motivation

Nowadays it is common for us to recognize printed texts and we also have no problem to learn to recognize our handwriting when we write all of the letters for machine learning training. Now we want to concentrate on historical handwritings. Would it not be amazing if we could only take a photo of a historical text in a museum and we would get the most important information about the text? Would it not be helpful for historians, when they find new handwriting, to be able to easily and quickly compare the text with all types of historical texts that exist in the whole world and see to which category it shall be classified and with what probability? Afterwards it will be easier for historians to concentrate on the text, century and probably also authors. We are deeply convinced that CNNs can help in the historical field of study.

1.2 Transfer learning

In classical supervised learning, the CNN learns weights using a dataset from the same domain and for the same task as we want to classify afterwards. On the contrary, in transfer learning [9] we have a source domain with a source task and during the learning we get the knowledge. We use this knowledge afterwards to solve another problem which is in some characteristics similar to the learning problem. Within transfer learning, we will concentrate on pre-training.

1.3 Data preprocessing

Mean subtraction, normalization and whitening are usually ranked among data preprocessing methods. Empricial mean and covariance matrix are usually computed for the training dataset and then applied on every single dataset (training, validation and testing). Instead, we will compute covariance from every single image and we will apply whitening over all pixels in one image. We will show that this is an effective technique. However, we can also preprocess the dataset in other ways. We could analyse where in the images lies the most important information. What is our main problem linked to a dataset during training? If it is the volume of data, which method should we use? We have two possibilities: (1) to pre-train the CNN or (2) to

widen the dataset by means of noise, rotation and translation of the images, cropping more random images and applying some additional transformations. Moreover, these methods make our learned model more robust. That is really important if we want to use it in real world afterwards.

1.4 State of the art

For the current direction of research, the best way is to look at the report from the ICFHR2016 Competition on the Classication of Medieval Handwritings in Latin Script [3]. The data from this report are listed in table 1.1, which shows the results in the same text type classification that we have. In the following subsections we will concentrate in detail on the solution of FAU and DeepScript.

Team	Accuracy	Method
FAU	0.8390	i-vector + SVMs
NNML	0.8380	CNN
FRDC-OCR	0.7980	CNN
DeepScript	0.7649	CNN VGG-architecture
TAU-3	0.5280	k-NN out of the training histograms
TAU-2	0.5010	k-NN out of the training histograms
TAU-1	0.4990	k-NN out of the training histograms

Table 1.1: Results from the ICFHR2016 competition

1.4.1 FAU

The FAU based the work on a strategy different from a CNN. They used i-vectors [10] and support vector machines (SVMs). At first, they modelled each category by the global distribution of feature vectors, which were calculated by means of i-vector extraction. Then, they applied within class covariance normalization (WCCN). The SVM has been used for classification. More detailed description is available in source [3].

1.4.2 Previous Work: DeepScript

The DeepScript implementation is important for us because the work was based on it. It uses the VGG architecture showed in table 3.1, the model uses a batch size of 30 images and stochastic gradient descent (SGD) with Nerestov momentum. Learning rate starts at 0.01 and every 10 epochs it is divided by 3. The training lasted for 30 epochs and used Keras with Theano as a backend.

The training dataset was divided into two subsets, training and validation, in the ratio $9 : 1$ for every group. Each image was downscaled with a factor of two and 100 random images were cut out. Then, each image was zoomed, rotated, translated etc. and at the end it was resized to the size of 150×150 pixels. In case of testing images, at first, they were downscaled, then 30 images of size 150×150 pixels were cropped from each image. And to set the final label, the predicted labels from all crops were averaged.

Chapter 2

Theoretical background

Basic information about CNNs are written in this chapter; firstly about linear classification, secondly about layers from which the neuronal network is composed. At the end of this chapter, backpropagation and optimizers are explained.

2.1 Neuron as a linear classificator

Before we speak about NN, we need to describe the basic parts that NN are made of. We need to speak about neurons [11]. A Neuron, also called perceptron, is a linear classificator. It means that the perceptron is a linear function that divides the space of inputs in two sections. The output of the perceptron function is computed by the following formula 2.1.

$$y = f\left(\sum_{i=1}^{n} (a_i \cdot w_i) + b\right) \tag{2.1}$$

Where b determines bias and for 2D we can imagine it as shift of a straight line. w_i are the weights, the parameters to be learned, in the 2D case they would represent the rotation. The function f represents the activation function (for example sigmoid, for details on this topic please refer to section 2.3). Classification output is marked in the formula with letter y. n denotes to the length of the input vector $\mathbf{a}$ and weight vector $\mathbf{w}$. For a better intuition please see the figure 2.1.

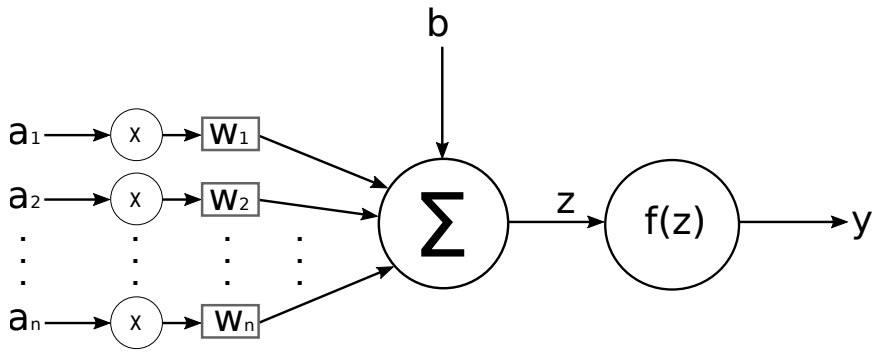


Figure 2.1: Basic scheme of a perceptron

2.2 Layers

A CNN is composed of layers and the layers are composed of neurons. The most important layers for our thesis are introduced in the following subsections. Before we get to architectures used in this thesis we need to create a general idea of CNNs. Figure 2.2 shows a simplified diagram of a CNN architecture ResNet50 with the corresponding input and output dimensions used for the CLaMM16 dataset. The input layer (marked by green colour in the diagram) must have the same number of neurons as the number of input pixels of one image and typically, the input layer is not counted to the number of layers of a CNN. Then, we can see hidden layers (marked by blue colour in the diagram) where the weights are trained. The dashed lines mark other hidden layers and their linkages. The hidden layers have different sizes, as we can see in the diagram, they have also different functions, which are described in following sections. The last layer, which is called output layer (marked by red colour in the diagram), gives us the result in which category the image will be classified. More detailed models of the CNNs used in this work are shown in the tables 3.1 and 3.2.

The using of layers have also the advantage, that the computations are matrix operations, therefore the calculations can effectively run in parallel on a graphics processing unit (GPU), where are the matrix computations usually faster than on a central processing unit (CPU).

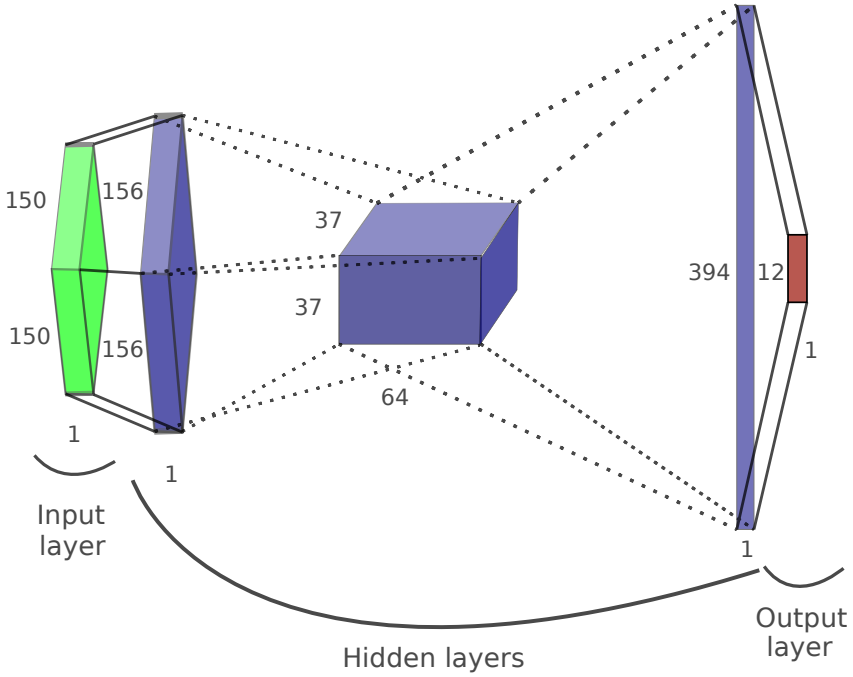


Figure 2.2: A simplified diagram of a CNN architecture ResNet50 with the corresponding input and output dimensions used for the CLaMM16 dataset

2.2.1 Input layer

The input layer does no computations, it only represents values. It is the first layer in the network and as such it is not counted to the number of CNN layers. Input size of this network corresponds to the size of the image, in our case it is (1, 150, 150), because we have a grayscaled images that have the width 150 and height 150 pixels. This example of input layer was mentioned before in the figure 2.2. If we would use RGB images then we could expect the size (3, image rows, image columns). In practical work, we need to be careful about compatibility of the dimension ordering of the network with the dimension ordering of input images.

2.2.2 Dense layer

The dense layer is fully connected, which was already used in the first types of NN. All inputs are connected with all outputs. The disadvantage is that there are a lot of weights to learn, the number of inputs is multiplied with

the number of outputs. Dense layers are often used as some of the last layers of the CNN, where the space has a lower dimension. Especially, the dense layer with softmax activation function is typically used as the last layer to normalize the output. For example the dense layer used in this thesis in ResNet50 in penultimate layer has 403,850 parameters, it comes from 1024 input parameters and 394 output parameters.

The dense layer is followed by an activation function, we used ReLU and Softmax in our work, described in more detail in sections 2.3.2 and 2.3.3. We can calculate the output of the dense layer according to equation 2.2.

$$o = f_a(\boldsymbol{A} \cdot \boldsymbol{W} + b) \tag{2.2}$$

Where o indicates an output of the dense layer, f_a determines the activation function, $\boldsymbol{A}$ is the input, then $\boldsymbol{W}$ represents weights and $\mathbf{b}$ determines bias. We can easily imagine calculation of one value of the output, if we look back at the figure of perceptron (2.1).

We will add here a concrete example, where we will use the ReLU activation function.

$$\boldsymbol{A} = \begin{matrix} 1 & -1 & -3 & 4 \end{matrix} \tag{2.3}$$

$$\boldsymbol{W}^T = \begin{matrix} 2 & 2 & 1 & 1 \\ 3 & 1 & 1 & -1 \end{matrix} \tag{2.4}$$

$$\boldsymbol{b} = \begin{matrix} 1 & 1 \end{matrix} \tag{2.5}$$

$$\boldsymbol{O} = \begin{matrix} 2 & 0 \end{matrix} \tag{2.6}$$

The values (2 ,0) are not a mistake because we also need to evaluate the influence of activation function on the result of the dot product (2 ,-4).

2.2.3 Pooling layer

Pooling layers are used to reduce the dimensions of the activation maps. The areas are defined by the pool size and stride length [7]. The pool size defines pooling windows of size $\mathbf{m} = (m_x, m_y)$ for 2D. Stride lengths $\mathbf{s} = (s_x, s_y)$ determine by how many pixels the window will be shifted. Eventually, a padding $\mathbf{p} = (p_y, p_x)$ can be added, if it is needed. When we designate the images as a matrix $\boldsymbol{A}$ with dimensions $\mathbf{a} = (a_x, a_y)$, we can write the output size $\mathbf{o} = (x, y)$ as stated in equations 2.7 and 2.8. It will be defined analogically for more dimensions.

$$x = \frac{a_x - (m_x - s_x) + 2 \cdot p_x}{s_x} \tag{2.7}$$

$$y = \frac{a_y - (m_y - s_y) + 2 \cdot p_y}{s_y} \tag{2.8}$$

This type of layers is used to reduce the number of parameters, the poooling layers have no parameters and can reduce the dimension of the image. Pooling layers also help to avoid overfitting. We will write in more detail about the max-pooling layer and average-pooling layer. However, there are other functions which can also be applied to the windows, almost all of them can be easily computed. We will not spend more time with these layers because we will not use them in architectures described in this thesis.

Max-pooling layer

The max-pooling finds out the maximum value in the window. In this case we have the max-pooling window size $\mathbf{m} = (2, 2)$ and the same size of stride lengths $\mathbf{s} = (2, 2)$. It means that in this case the size will be reduced to one quarter, illustratively the input to max-pooling layer is matrix 2.9 and the output can be seen in matrix 2.10.

$$\begin{array}{cc|cc} 2 & 3 & 5 & 3 \\ 4 & 7 & 8 & 1 \\ \hline 3 & 3 & 9 & 8 \\ 4 & 2 & 4 & 7 \end{array} \tag{2.9}$$

$$\begin{array}{cc} 7 & 8 \\ 4 & 9 \end{array} \tag{2.10}$$

Average-pooling layer

The average-pooling layer computes the average of all pixels in the window. As an example, we will calculate it for matrix $\boldsymbol{A}$ with size (5,4); other parameters are average-pooling window $\mathbf{m} = (3, 3)$ and stride lengths $\mathbf{s} = (3, 2)$. It means that in this case the output size will be reduced to 2×2, illustratively, the input to average-pooling layer, after padding $\mathbf{p} = (1, 0)$ was applied, is shown in matrix 2.11 and the output can be seen in matrix 2.12. Zeropadding adds one column with zeros to the left side and to the right side.

$$\begin{array}{ccc|ccc} 0 & 1 & 2 & 5 & 3 & 0 \\ 0 & 2 & 7 & 8 & 1 & 0 \\ 0 & 3 & 3 & 9 & 8 & 0 \\ 0 & 4 & 2 & 4 & 7 & 0 \end{array} \tag{2.11}$$

$$\begin{array}{cc} 2.0 & 3.\overline{7} \\ 2.\overline{3} & 4.\overline{1} \end{array} \tag{2.12}$$

2.2.4 Convolutional layer

Convolutional layers [11] are the basic building units of CNNs. They try to find some specified property using the filters. When we have an input with large dimension (for example an image) and we use fully connected layer, then we get a lot of parameters. However, when we use convolutional layer, then we can benefit from the parameter sharing (weight sharing) and we have relatively few parameters in comparison with fully connected layer, which is one of the biggest advantages of convolutional layers. Parameter sharing represents sharing of the weights spatially across every single filter type.

Moreover, the receptive field is a local connectivity of a neuron to the input volume. The receptive field indicates the size of the filter and is computed over the depth of input to the convolutional layer. That is the reason, why the filter size 1×1 is meaningful. It is a convolution of $1 \times 1 \times depth$, also used in our work. Similar depth of input and filter has furthermore positive influence on number of parameters.

Second and even more important property of parameter sharing is that it prevents the overfitting, because it is more important that the shape occurs in the input to convolutional layer than where the concrete shape or another feature that is filtered is. The first several convolutional layers are more general and look for basic shapes, for example edges, however the several last convolutional layers look for more complex shapes.

As was already computed for the pooling layer, it is also easily possible to determine an output size for a convolutional layer. The output size is defined by a windows size $\mathbf{m} = (m_x, m_y)$ for 2D. Stride lengths $\mathbf{s} = (s_x, s_y)$ define by how many pixels the window will be shifted. Eventually, padding $\mathbf{p} = (p_y, p_x)$ can be added if it is needed. When we define the image as matrix $\boldsymbol{A}$ with dimensions $\mathbf{a} = (a_x, a_y)$ and we determine k as the number of filters, then, we can write the equations for outputs, equation 2.13 for width, equation 2.14 for height and equation 2.15 for depth, as listed bellow.

$$x = \frac{a_x - m_x + 2 \cdot p_x}{s_x} + 1 \tag{2.13}$$

$$y = \frac{a_y - m_y + 2 \cdot p_y}{s_y} + 1 \tag{2.14}$$

$$d = k \tag{2.15}$$

Moreover, it is needed to calculate the number of parameters. There are two types of parameters: The learnable weight matrix $\boldsymbol{W}$ and the bias $\mathbf{b}$. Bias is sometimes subsumed to $\boldsymbol{W}$ under the label $\mathbf{w}_0$. Letter k stands for the number of filters, variable d_0 is input depth and variable $m = (m_x, m_y)$ stands for the size of window (here also the size of filter). Then, we can calculate the number of shared parameters as specified in equation 2.16.

$$NP = (m_x \cdot m_y \cdot d_0) \cdot k + k \tag{2.16}$$

The third piece of information that we want to get from the convolutional layer is the output volume $\mathbf{o} = (o_x, o_y)$. We can calculate it for d-th depth and concrete window $\boldsymbol{M}$ filtered with filter $\boldsymbol{F}$ of size $\mathbf{f} = (f_x, f_y)$. The equation for convolution is mentioned in equation 2.17.

$$o_{x,y,d} = \left(\sum_{i=1}^{f_x} \sum_{j=1}^{f_y} f_{j,i,d} \cdot a_{(j+x),(i+y),d} \right) + b_d \tag{2.17}$$

To easily understand convolutional network we add here an example for a 5×5 input matrix and for two stride lengths $\mathbf{s} = (2, 2)$ and padding $\mathbf{p} = (0, 0)$. The input is seen in matrix 2.18, the filter is written in matrix 2.19, the output is shown in matrix 2.20 and bias is $b = 2$.

$$\boldsymbol{I} = \begin{matrix} 1 & 3 & 1 & 0 & -1 \\ -1 & 2 & 0 & 6 & 1 \\ -1 & -6 & 0 & 0 & -1 \\ 4 & -1 & 3 & 2 & 1 \\ -1 & 2 & 0 & 2 & -3 \end{matrix} \tag{2.18}$$

$$\boldsymbol{F} = \begin{matrix} -1 & 0 & -1 \\ -1 & 0 & -1 \\ -1 & 0 & -1 \end{matrix} \tag{2.19}$$

$$\boldsymbol{O} = \begin{matrix} 2 & 2 \\ -3 & 2 \end{matrix} \tag{2.20}$$

We used the filter $\boldsymbol{F}$ for convolution, we show the calculation for a convolution depicted in blue by matrix 2.18 with the filter $\boldsymbol{F}$ and the result can be found in matrix 2.20 in the blue cell. Calculation 2.21 according to equation 2.17 follows as:

$$o(1,2,1) = \left(\sum_{i=1}^{3} \sum_{j=1}^{3} f_{j,i,1} \cdot a_{(j+2),(i+2),1} \right) + b_1 = (-1+1-1+1)+2 = 2. \tag{2.21}$$

2.2.5 Dropout layer

The most important property of *dropout* is that it helps against overfitting. With a determined probability, this layer interrupts the connection to the next layer and gives us the possibility to learn the network in more ways, which makes the network more general. The dropout layer is used only during the train time and has no effect during the testing time [12], because we want deterministic output during the testing time. Simultaneously, the neurons are only multiplied by the dropout factor to get the same activation range for the subsequent layers. The biggest question related to this kind of layer is which probability to choose for dropout. Dropout probability 0.5 is commonly used and it is also in the network architecture VGG16. The probability in dropout layers has usually the same value over the entire network [11].

2.2.6 Batch normalization layer

At first, we will define internal covariate shift according to Sergey Ioffe and Christian Syengedy, *"We define Internal Covariate Shift as the change in the distribution of network activations due to the change in network parameters during training."* [14] If the internal covariance shift is small, then the training progresses faster. The internal covariance shift can be reduced by a normalization of the input pictures (whitening, see section 5.2) or through the batch normalization. The batch normalization layer normalizes the activations close to the standardized normal distribution for every batch [7]. The mean value of activation is approximately zero and the standard deviation is almost one. The normalization is

$$\hat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}. \tag{2.22}$$

Where $\hat{x}^{(k)}$ is the normalized kth dimension of input, $x^{(k)}$ denotes the kth dimension of the input, $\mathrm{E}[x^{(k)}]$ represents the mean value of $x^{(k)}$ and Var expresses the variance of $x^{(k)}$. Moreover, the batch normalization allows the network to scale and shift the normalization with parameters γ and β, which are parameters of linear function 2.23.

$$y^{(k)} = \gamma^{(k)} \cdot \hat{x}^{(k)} + \beta^{(k)} \tag{2.23}$$

The batch normalization layer can be found after the fully-connected layer or after the convolutional layer. The batch normalization is before the activation function, because the non-linearity of the activation function would cancel the positive influence of batch normalization on the internal covariance shift.

The batch normalization layer learns the parameters γ and β during backpropagation and they are learned per feature map. Parameters γ and β are different for every dimension. The mean value E and variance Var are not calculated during the testing. However, the mean value and the variance from the training set are used.

Moreover, this layer restricts dependency of the final results on initializations of weights. Furthermore, when we use batch normalization, then we can set up a higher learning rate. More information about batch normalization and how to code it is available in paper [14].

2.3 Activation functions

Activation functions [11] are important, because they add nonlinearity to the NN. We will concentrate on ReLU and Softmax, because they are also used in ResNet50 and VGG16. For all activation functions, x_i represents one sample from input to activation function and y_i determines output from the activation function for one sample.

2.3.1 Historical activation functions

Two main activation functions mostly used in history are sigmoid, shown in equation 2.24, and hyberbolic tangent (tanh), shown in equation 2.25. Output of the sigmoid function is located between zero and one. Output of the tanh takes value in the range of $(-1, 1)$. We can find three disadvantages of sigmoid. Firstly, if the gradient is very small, it will be almost zero after passing through sigmoid function. Secondly we need to choose initial values carefully not to get into saturation. The third issue is that sigmoid outputs are positive or zero all the time. It causes that all gradients will be positive or all gradients will be negative during the backpropagation. It can bring instability to the network and a consistent changing between all positive and all negative gradients. Conversely, tanh is zero-centered, therefore, it is more popular than the sigmoid. We can compare both functions in figure 2.3.

$$y_i = \frac{1}{1 + \exp(-x_i)} \tag{2.24}$$

$$y_i = \tanh(x_i) \tag{2.25}$$

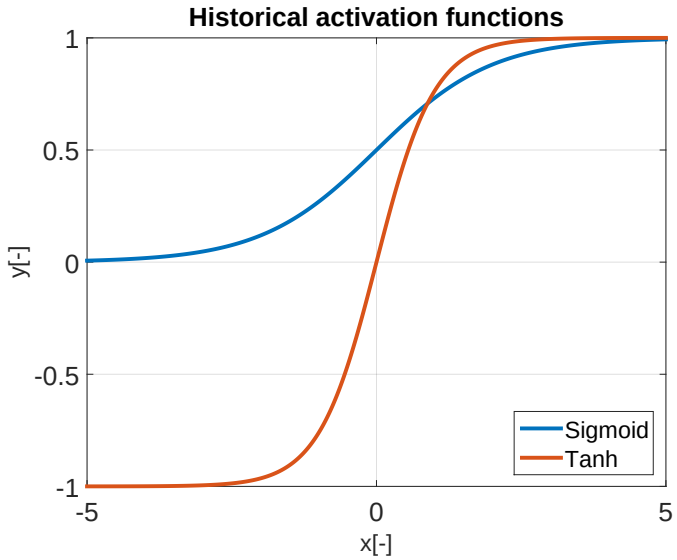


Figure 2.3: Sigmoid and tanh activation functions

2.3.2 Rectified Linear Unit

ReLU [11], shown in equation 2.26 and drown in figure 2.4, is currently very popular. ReLU has two main advantages, it accelerates convergence in comparison with other activation functions and the second advantage is computational unpretentiousness. On the other hand, a big disadvantage is that ReLU can cause "dying" of neurons, it means that the output value is set to zero from a specific moment till the end of learning. It can be caused by a big input value, when the big value goes throw a ReLU, it influences changing of weights in such way that the neuron will never activate on any data. This phenomenon is mostly caused by setting too high learning rate.

$$y_i = \max(x_i, 0) \tag{2.26}$$

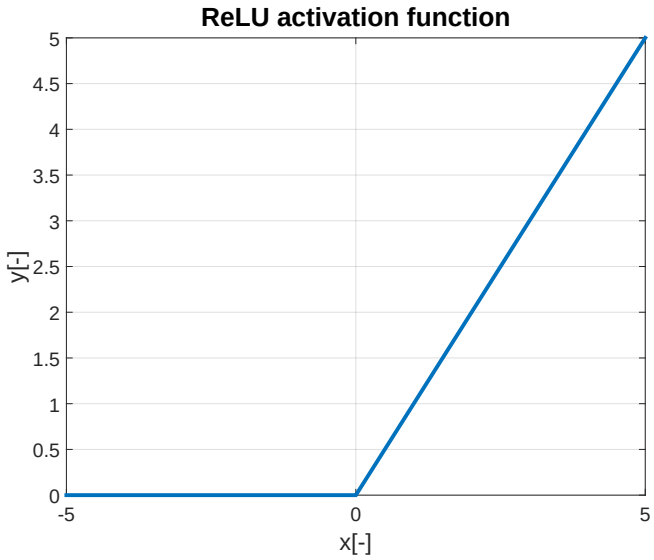


Figure 2.4: ReLU activation function

We can meet many types of ReLU. For example the leaky ReLU, which solves the problem of "dying" neurons. If the input to the leaky ReLU is less than zero, then the output value is the input value multiplied by a small positive constant (α) instead of being zero. The output value is computed according to equation 2.27.

$$y_i = \max(x_i, \alpha \cdot x_i) \tag{2.27}$$

2.3.3 Softmax

As we can see from equation 2.28, output from softmax [15] has to be positive numbers or zero all the time. If we divide the exponential of input to softmax y_i by the sum of all exponentials of inputs to softmax, we get the probability for classification to class i. We find the advantage of Softmax in the possibility of application on datasets with many categories.

$$P(y_i|x_i; W) = \frac{e^{y_i}}{\sum_j e^{y_j}} \tag{2.28}$$

Function P (2.28) is the probability assigned to the label y_i for image x_i where W represents parametrization [11]. Softmax activation function is often used in the last layer to normalize the values so that the final sum of output values equals one.

2.4 Backpropagation

Before we get to the backpropgation, we will outline what is the forward pass. The forward pass is a computation of all classes scores with respect to weights and then we compute the cost function. This cost function will be then minimized during the backpropagation. The backpropagation [11] is a process to calculate the gradient of the specified function on the input vectors. To simplify and accelerate the computation, it is needed to use the chain rule, described by equation 2.29, for computing the influence of the input on the loss (output).

$$\frac{\partial o}{\partial x} = \frac{\partial o}{\partial z} \cdot \frac{\partial z}{\partial x} \tag{2.29}$$

Where o represents the output function, x is the variable for which we want to calculate the weights and z determines a subfunction from function o. Inputs and outputs are defined from the perspective of forward propagation.

The elementary functions in CNNs can be imagined as gates. For example, addition distributes current gradient. Maximum function chooses the maximum from all inputs and writes the gradient to selected input, it writes zero to the other inputs because they have no effect on the output. When we speak about a multiplying gate, it multiplies all of the inputs, except the one which it is counted for, with the gradient. When we have one input and multiple outputs, then the gradients from different branches are summed during the backpropagation.

It is needed to remember the weights from the forward pass for computations in the backward pass. The local gradients are usually expressed in Jacobian of input vectors. More informations about the parameters updates are available in following section 2.5.

2.5 Optimizer

The optimizer, also called optimization algorithm, is an algorithm to minimize the loss function. In general it is the way of learning the parameters of the network $\boldsymbol{\theta}$, for example weights, biases and parameters for batch normalization layers (shift and scale). A really important hyperparameter for the optimizer is the learning rate, which indicates the degree of change and which also influences the learning speed.

We will focus on gradient descent in this section. We have three basic possibilities how to calculate the gradient descent: batch gradient descent, stochastic gradient descent and mini-batch gradient descent. At the end of this section we will write about additional features for quicker optimization. An important source for this and next four sections is the overview by Sebastian Ruder [17], where more informations about the subject can be found.

2.5.1 Batch gradient descent

Batch gradient descent (BGD) computes the gradient for the whole dataset and updates the parameters at once. BGD has two disadvantages. Firstly, the computation of a big gradient takes a lot of time and we get no output during the computation and secondly, we need to have enough memory space to load the whole dataset into it. Conversely BGD will converge to a global minimum, if it is a convex function, otherwise it will end in a local minimum. How the model parameters $\boldsymbol{\theta}_t$ are updated is written in equation 2.30.

$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \eta \cdot \nabla_{\boldsymbol{\theta}_{t-1}} \cdot \boldsymbol{J}\left(\boldsymbol{\theta}_{t-1}\right) \tag{2.30}$$

Where $\boldsymbol{\theta}_{t-1}$ represents the parameters before the updating, $\boldsymbol{J}\left(\boldsymbol{\theta}_{t-1}\right)$ determines an objective function (loss function) and the letter η indicates the above mentioned learning rate.

2.5.2 Stochastic gradient descent

SGD computes the gradient gradually for each image (sample from the training dataset). How the model parameters $\boldsymbol{\theta}_t$ are updated is written in equation 2.31.

$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \eta \cdot \nabla_{\boldsymbol{\theta}_{t-1}} \cdot \boldsymbol{J}\left(\boldsymbol{\theta}_{t-1}, x^i, y^i\right) \tag{2.31}$$

Where $\boldsymbol{\theta}_{t-1}$ represents the parameters before the updating, $\boldsymbol{J}\left(\boldsymbol{\theta}_{t-1}, x^i, y^i\right)$ determines an objective function (loss function) for one sample x^i with label y^i. The letter η indicates the learning rate.

2.5.3 Mini-batch gradient descent

The last but not least variant is using mini-batch gradient descent (MBGD). The MBGD was also used to optimize the models ResNet50 and VGG16. This

type of optimizer is more fluent during optimization than SGD because MBGD iterates over the batches. The batch size is 30 images in our case. Moreover, it is also quicker in comparison with BGD. How the model parameters $\boldsymbol{\theta}_t$ are updated is written in equation 2.32.

$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \eta \cdot \nabla_{\boldsymbol{\theta}_{t-1}} \cdot \boldsymbol{J}\left(\boldsymbol{\theta}_{t-1}, x^{(i,i+n)}, y^{(i,i+n)}\right) \tag{2.32}$$

Where $\boldsymbol{\theta}_{t-1}$ represents the parameters before the updating, $\boldsymbol{J}\left(\boldsymbol{\theta}_{t-1}, x^{(i,i+n)}, y^{(i,i+n)}\right)$ determines an objective function (loss function) for mini-batch of samples $x^{(i,i+n)}$ with labels $y^{(i,i+n)}$. The number of samples in one batch is labelled by the letter n. The symbol η indicates the learning rate.

2.5.4 Additional techniques

We will write about two techniques: Firstly, about momentum that helps to get to the minimum and prevents oscillation. Secondly, we will describe how Nesterov accelerated gradient (NAG) works, it helps not to overshoot. Both of these additional features were also used for optimization in this work.

At first, we will write about momentum. The usual value of momentum is 0.9, which is also used in our case. If we add influence of last added parameters (θ_{t-1}), then we make changing the direction harder. This also speeds up the optimization. The equations for computation of the actual parameters using momentum are:

$$\mathbf{v}_t = (\gamma \cdot \mathbf{v}_{t-1} + \eta \cdot \nabla_{\boldsymbol{\theta}_{t-1}} \cdot \boldsymbol{J}(\boldsymbol{\theta}_{t-1})), \tag{2.33}$$

$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \mathbf{v}_t. \tag{2.34}$$

Where $\boldsymbol{\theta}_{t-1}$ represents the parameters before the updating, $\boldsymbol{J}(\boldsymbol{\theta}_{t-1})$ determines an objective function (loss function). The letter η indicates the learning rate, $\mathbf{v}_{t-1}$ is the last vector update and γ determines momentum term.

Secondly, we will describe NAG. The NAG first does a big step in the direction of the last step, after that it computes the actual gradient and then it compares them and corrects them by adding a correction vector. The formulas for computation of the actual parameters using momentum are:

$$\mathbf{v}_t = \gamma \cdot \mathbf{v}_{t-1} + \eta \cdot \nabla_{\boldsymbol{\theta}_{t-1}} \cdot \boldsymbol{J}(\boldsymbol{\theta}_{t-1} - \gamma \cdot \mathbf{v}_{t-1}), \tag{2.35}$$

$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \mathbf{v}_t. \tag{2.36}$$

Where $\boldsymbol{\theta}_{t-1}$ represents the parameters before the updating, $\boldsymbol{J}(\boldsymbol{\theta}_{t-1})$ determines an objective function (loss function). The letter η indicates the learning rate. The $\mathbf{v}_{t-1}$ is the last vector update, $\mathbf{v}_t$ represents the actual vector update and γ refers to the momentum term.

Chapter 3

Model setup and training

In this chapter we will describe the solution using a CNN system for script style classification. We will outline the basis of our implementation and which main libraries we used. Then, we will write about supervised learning and used models.

3.1 Basis

We have decided to base the Bachelor's thesis on existing libraries and scripts. We based the implementation on the code from Mike Kestemont which he used for the ICFHR2016 competition [6]. We have used Keras and Python since Mike Kestemont's code is based on that.

3.2 Important libraries

We have used Keras using Tensorflow [8] as backend. Tensorflow is an open-source library for numerical computation. We selected the GPU version, which is based on CUDA[1] allowing a much faster computation than employing the CPU.

[1] `https://medium.com/@acrosson/installing-nvidia-cuda-cudnn-tensorflow-and-keras-69bbf33dce8a`

3.3 Supervised learning

In computer vision we have four basic machine learning specializations: supervised learning, transfer learning, unsupervised learning and reinforcement learning. Most known types of supervised learning are SVM, k nearest neighbors (kNN), decision trees and NN. According to Bishop, supervised learning is defined as an *"application in which the training data comprises examples of the input vectors along with their corresponding target vectors are known as supervised learning problems."*[16]

We train the network on a training dataset and then we test it on a testing dataset. As we want to simulate real contest conditions, we do not want to get access to the testing dataset until we finish training, but on the other hand we need to avoid overfitting. Therefore we split the train dataset into the training dataset and the validation dataset. The validation dataset ensures that we have a dataset independent of the training dataset, on which we can test accuracy during the training. We save the model with weights only if the validation accuracy improves against the last saved model.

We get the dataset for the competition already split into training and testing parts in the ratio 2:1. Because of that we need to split only the training dataset into validation and training datasets. For example we split CLaMM16 in the ratio 9:1 (training:validation). A small part of the training dataset is usually separated for the validation dataset, especially, if the dataset is relatively small, as in our case. We can read more about used datasets in sections 4.2 and 4.3.

3.4 Model and sequential model

The sequential model [7] is based on the list of layers that we add to the model. This means that we have to choose the whole NN architecture when we are preparing the model. The most important layers are convolutional, activation, max-pooling, dropout and batch normalization layers, more information about layers are available in chapter Theoretical background in section 2.2.

However, at first, it is better to use an established architecture such as VGG16 [1] or ResNet50 [2]. We can see an example of the VGG16 architecture in table 3.1. This architecture is used in this work for CLaMM16 and has

87,879,500 trainable parameters.

The ResNet50 architecture is shown in table 3.2 and the repetitive parts of the architecture are shown in the convolutional block table 3.3 and identity block table 3.4. The model ResNet50 uses an input size (1, 150, 150) and 12 output classes. In total it has 6,351,628 parameters, where 6,327,180 of them are trainable and 24,448 of them are non-trainable.

Function	Output shape	Parameters
ZeroPadding2D	(1, 152, 152)	0
Convolution2D	(64, 150, 150)	640
ZeroPadding2D	(64, 152, 152)	0
Convolution2D	(64, 150, 150)	36,928
MaxPooling2D	(64, 75, 75)	0
ZeroPadding2D	(64, 77, 77)	0
Convolution2D	(128, 75, 75)	73,856
ZeroPadding2D	(128, 77, 77)	0
Convolution2D	(128, 75, 75)	147,584
ZeroPadding2D	(128, 77, 77)	0
Convolution2D	(128, 75, 75)	147,584
MaxPooling2D	(128, 37, 37)	0
ZeroPadding2D	(128, 39, 39)	0
Convolution2D	(256, 37, 37)	295,168
ZeroPadding2D	(256, 39, 39)	0
Convolution2D	(256, 37, 37)	590,080
ZeroPadding2D	(256, 39, 39)	0
Convolution2D	(256, 37, 37)	590,080
MaxPooling2D	(256, 18, 18)	0
Flatten	(82944)	0
Dense	(1024)	84,935,680
Dropout	(1024)	0
Dense	(1024)	1,049,600
Dropout	(1024)	0
Dense	(12)	12,300

Table 3.1: VGG16, for input size (1, 150, 150) and 12 classification categories

Function	Output shape	Parameters
InputLayer	(1, 150, 150)	0
ZeroPadding2D	(1, 156, 156)	0
Convolution2D	(64, 75, 75)	3200
BatchNormalization	(64, 75, 75)	0
Activation	(64, 75, 75)	0
MaxPooling2D	(64, 37, 37)	0
convolutional block	(256, 37, 37)	39,744
identity block	(256, 37, 37)	71,552
identity block	(256, 37, 37)	71,552
convolutional block	(512, 19, 19)	383,232
identity block	(512, 19, 19)	282,368
identity block	(512, 19, 19)	282,368
identity block	(512, 19, 19)	282,368
convolutional block	(1024, 10, 10)	1,520,128
identity block	(1024, 10, 10)	1,121,792
identity block	(1024, 10, 10)	1,121,792
identity block	(1024, 10, 10)	1,121,792
AveragePooling2D	(1024, 1, 1)	0
Flatten	(1024)	0
Dense	(12)	12,300

Table 3.2: ResNet50, for input size (1, 150, 150) and 12 classification categories

Function	Output shape	Parameters
Convolution2D	(a, b, b)	$a \cdot a + a$
BatchNormalization	(a, b, b)	$4 \cdot a$
Activation	(a, b, b)	0
Convolution2D	(a, b, b)	$3 \cdot 3 \cdot a \cdot a + a$
BatchNormalization	(a, b, b)	$4 \cdot a$
Activation	(a, b, b)	0
Convolution2D	$(4 \cdot a, b, b)$	$4 \cdot a \cdot a + a$
Convolution2D	$(4 \cdot a, b, b)$	$4 \cdot a \cdot a \cdot s + a$
BatchNormalization	$(4 \cdot a, b, b)$	$4 \cdot 4 \cdot a$
BatchNormalization	$(4 \cdot a, b, b)$	$4 \cdot 4 \cdot a$
Merge	$(4 \cdot a, b, b)$	0
Activation	$(4 \cdot a, b, b)$	0

Table 3.3: ResNet50, convolutional block with input size $(a \cdot s, s \cdot (b-1)+1, s \cdot (b-1)+1)$, where s is stride length

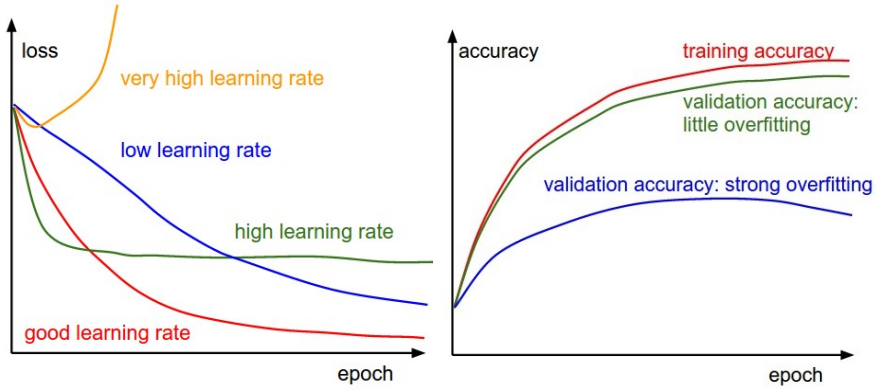
Function	Output shape	Parameters
Convolution2D	$(a/4, b, b)$	$a \cdot a/4 + a/4$
BatchNormalization	$(a/4, b, b)$	$4 \cdot a/4$
Activation	$(a/4, b, b)$	0
Convolution2D	$(a/4, b, b)$	$3 \cdot 3 \cdot a/4 \cdot a/4 + a/4$
BatchNormalization	$(a/4, b, b)$	$4 \cdot a/4$
Activation	$(a/4, b, b)$	0
Convolution2D	(a, b, b)	$a \cdot a/4 + a$
BatchNormalization	(a, b, b)	$4 \cdot a$
Merge	(a, b, b)	0
Activation	(a, b, b)	0

Table 3.4: ResNet50, identity block with input size (a, b, b)

Firstly, it is needed to build the architectures. Secondly, we need to compile the models in Keras. The used error metric is customarily accuracy. Then, we start to train. Here, it is important to consider how long we want to train and if we want to do episodic saving. The advantage of episodic saving is the ability to resume training of a specific model instant, for example, on another dataset. This is also described in this thesis in section 4.3.2 and it is implemented in the attached script. At the end, we test the learned model. If you want to try the learning and testing of the CNN on your own, then please follow the steps in Appendix B.

3.5 Learning rate adjustment

During the training it is important to observe tendencies of learning because we do not want to let an unsuitable script run for a long time We want to stop the learning early enough before it starts to overfit. For example, we can observe a trend of changing the loss function, the training and the validation accuracy. We can also observe the concrete numbers, or for easier monitoring, we can draw a graph of accuracy or loss. An example of a validation accuracy graph for our data is in figure 5.3. More theoretically, the shapes of these graphs can be various but in a simplified example, we can imagine tendencies as drown in figures 3.1a and 3.1b.



(a) : Learning rates [18] **(b) :** Accuracies [19]

Figure 3.1: Observed parameters

Another possibility is to calculate the ratio of updated weights or gradient distribution per layer. The last but not least is the possibility to visualize the first convolutional layer. If it is smooth and consists of diverse features, then the learning is likely proceeding correctly [11]. The last three possibilities have not been used in our work.

3.6 Initialization of weights and hyperparameters

At first, we will discuss the weights. It might seem as a good idea to initialize all the weights to zero. However, it is not because then all neurons in the same layer will have the same output. That also means the same weight update. Biases are normally initialized to zero. In our case, we will use two different initializations. We will initialize the weights to small positive and negative random numbers and also to pre-trained weights [11].

For CNNs it is really important to initialize the learning rate to an appropriate number. Because if we choose a too big learning rate, then we risk that we will miss the minimum. If we choose a too low learning rate, then it will take a very long time to get to the minimum and it is more probable that it will end in a local minimum that is much bigger then the other local minima. In our case, the best results for the CLaMM16 dataset without whitening were brought by the learning rate of 0.01. After that we tried to change this hyperparameter also during fine-tuning, we can read the results in section 4.3.2.

Chapter 4

Usefulness evaluation of the pre-training

In this chapter we will concentrate on the evaluation of pre-training for script type classification using other datasets. Figure 4.1 shows a diagram illustrating the datasets and preprocessing techniques we have used.

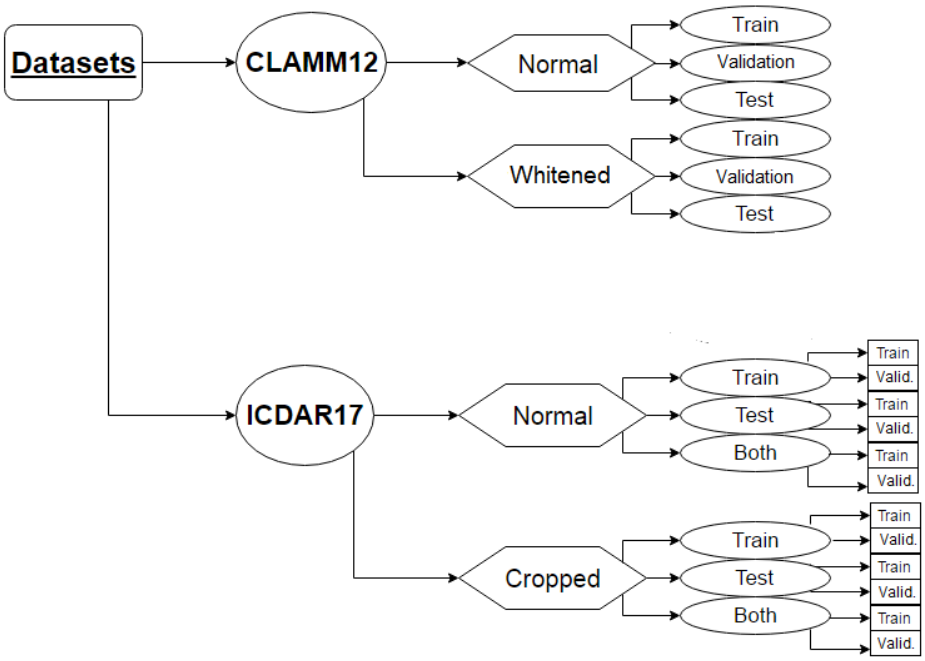


Figure 4.1: Diagram for used datasets

4.1 Introduction

One of the most effective techniques to improve the testing error, when we have a small training dataset, is the use of another dataset for pre-training. At first, we learn a CNN on the pre-training data. Alternatively, we can also use an already pre-trained network, which we can find in the library Caffe (Model ZOO[1]). However, we have not chosen this option but we have pre-trained the network on our own. Afterwards, we removed the last fully connected (FC) layer, we added another FC which has the correct number of output classes (in our case 12) and we fine-tuned it on other data. During the training, we visualized the validation error (see for example Figure 4.4). It is also recommended to visualize the convolutional (CONV)/FC filters (matrix W). We have decided to use the ICDAR17 dataset for pre-training.

4.2 CLaMM16

CLaMM16 is the dataset used for the competition "ICFHR2016 [5] Competition on the Classification of Medieval Handwritings in Latin Script". The dataset consists of images from 12 categories: caroline, cursiva, half-uncial, humanistic, humanistic-cursive, hybrida, praegothica, semihybrida, semitextualis, southern-textualis, textualis and uncial. The testing dataset includes 1000 images and the training dataset contains 2000 images. We decided to split the training dataset into a training set and a validation set using the ratio of 9:1, it means for the dataset CLaMM16, there are 1800 training images and 200 validation images. The validation set is used to counter overfitting. Overfitting means that the learning adapts too much to the training data and does not generalize on unseen test data. The size of input images for the CNN was chosen to be 150x150 pixels, where whole words can be seen, we can find an example in figure 4.2.

Figure 4.2: Example of CLaMM16 image in size of input to the CNN

[1] https://github.com/BVLC/caffe/wiki/Model-Zoo

Preprocessing test data is available in a file structure on CD[2]. For the validation of our work, we use the test dataset.

We let the training run for 25 epochs, but we plot only the first 10 epochs because we achieved the best accuracy of the validation dataset for ResNet50 4.4 in the 7th epoch and for VGG16 4.3 in the 4th epoch. The training runs with SGD optimizer and with learning rate 0.01, which changes every 10 epochs to half, but it has no influence, because we achieved the best accuracy in the first 10 epochs in both cases. Our augmented dataset for training has 180,000 samples and we used the batch size of 30 samples, which is 6000 parameters updates each epoch. Our augmented dataset for testing has 30,000 samples, from which the classification for 1000 images was predicted, which is 30 samples to predict one image. You can compare concrete numbers of validation accuracy and testing accuracy[3] in table 4.1.

Model	Validation accuracy	Test accuracy
VGG16	0.895	0.764
ResNet50	0.905	0.799

Table 4.1: CLaMM16 accuracy

[2] `\program\DeepScript-master\data\all\CLaMM16\test\`

[3] There is more to see in directory `\program\DeepScript-master\models\`, then choose directory CLAMM12BasicResnet50 or CLAMM12BasicVgg16 and then look in files test.out and train.out.

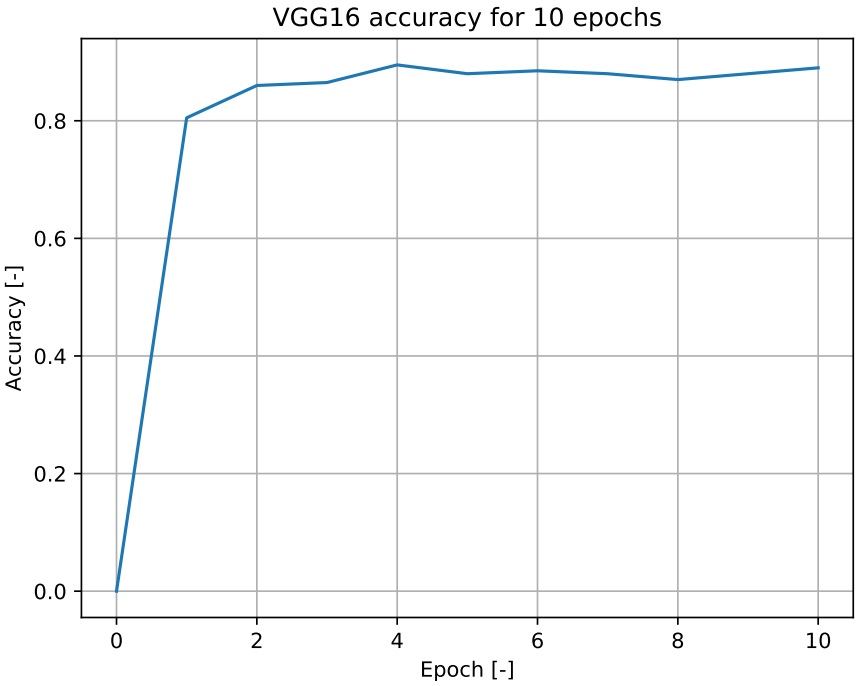


Figure 4.3: Validation accuracy of VGG16

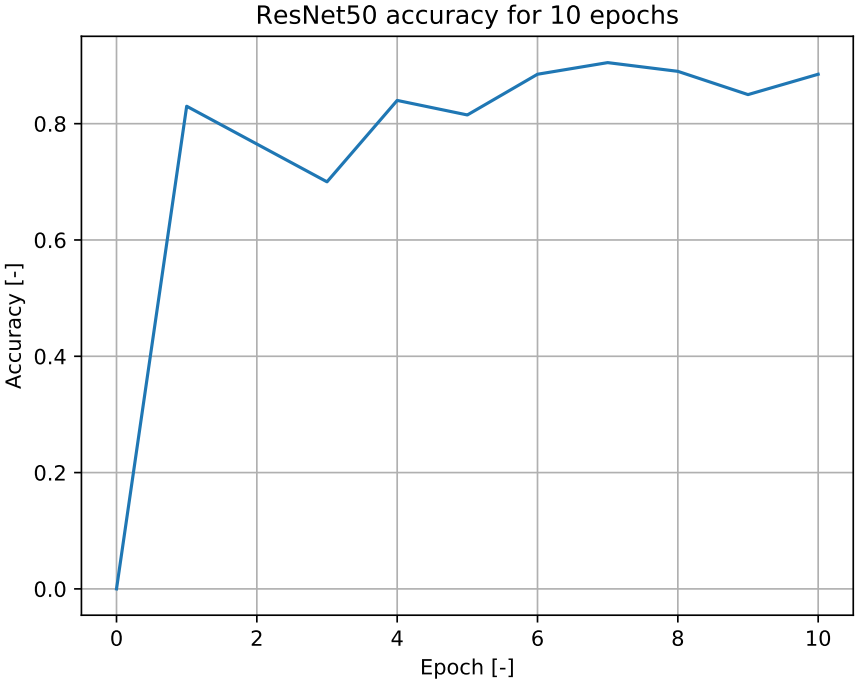


Figure 4.4: Validation accuracy of ResNet50

We suppose that ResNet50 has better accuracy because it is deeper and it can cover more details. Resnet50 also contains batchnormalization layers, which has a similar function as whitening (5.2) which yields good results. Batchnormalization layers help to solve the problem of different distributions of features across the batches. The issue is internal co-variate shift, which represents that even small changes of weights in the first several layers of large networks can have a big influence on weights of subsequent layers. That could destroy the already learned features, which is not desirable.

Batch normalization also helps with outliers at the start of the learning [14]. More about batchnormalization and explanation of how it works can be found in section 2.2.6. Because we have got better accuracy for the ResNet50 model during the training on CLaMM16, we have decided to concentrate on ResNet50 for pre-training.

4.3 ICDAR17

Each category in the dataset comes from a different writer. The task is a writer classification. All of the handwritings come from the 13th to 20th centuries. In most cases the texts are written in Latin, German and French. The type of handwritings is letters [20], which differs from the CLaMM16, where there are mostly parts of books. The images in ICDAR17 are sized and scaled differently. Moreover, the text on images is scattered differently, which is a disadvantage for NN learning when we use random cropping of the image.

Firstly, we split up the dataset into two different files, secondly we cropped eight areas of 224x224 pixels from the specific area of the input image, because we need to learn from text areas and not blank edges, because the text area contains most of the information about the writer. This area and also the cut-outs are specified by equations 4.1 and 4.2. In the end, we need to augment the dataset.

$$l = round\left(\frac{w_0 - CNTW \cdot w_1}{k_l}\right) + x \cdot w_1 \tag{4.1}$$

Where l represents the x coordinate of the top left corner of the cropped image, w_0 is the width of original image, w_1 determines the width of cut-out, x expresses the cut-out index in x direction, $CNTW$ stands for the total number of cut-outs in axis x, $round$ is rounding to whole numbers and k_l indicates a ratio of left and right edges.

$$t = round\left(\frac{h_0 - CNTH \cdot h_1}{k_t}\right) + y \cdot h_1 \tag{4.2}$$

Where t represents the y coordinate of the top left corner of the cropped image, h_0 is the height of original image, h_1 determines the height of cut-out, y expresses the cut-out index in y direction, $CNTH$ stands for the total number of cut-outs in axis y, *round* is rounding to whole numbers and k_t indicates a ratio of top and bottom edges.

In our case, the constant $k_l = 1.5$ indicates a ratio of left and right edges 2:3 and the constant $k_t = 5$ indicates a ratio of top and bottom edges 1:5. We use this ratio instead of fixed position because the images have different size and scale. This type of cropping is more reusable. Another possibility is to rescale or to resize all images before cropping but this brings a loss in image quality, which can worsen the CNN learning.

Moreover, it was needed to treat the condition for cropping only inside the image because the function pillow adds pixels outside the image with black colour. In the end, the images were changed from the color model (RGB) to grayscale to be more similar to the CLaMM16 dataset.

In the figure 4.5, we can see parts that were cropped from the image delimited by the black contours.

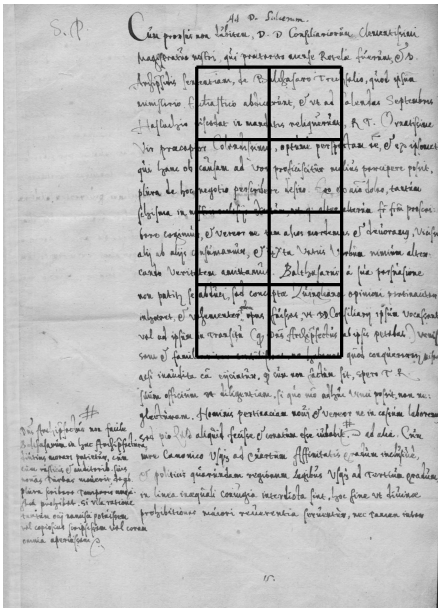


Figure 4.5: Scheme for cropping document $CNTW = 2$, $CNTH = 4$

After that we modified images in the same way as done in the CLaMM16 dataset. In this case, we only need to split the images in training and validation datasets and grayscale them. An example of a image edited in this

way can be seen in the following figure 4.6.

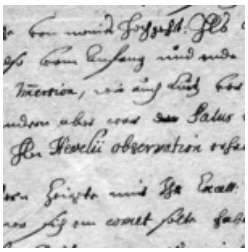


Figure 4.6: Edited image from ICDAR17 for normal dataset

We used the file format tiff, in uncompressed version, as it is the same that is used by CLaMM16.

4.3.1 ICDAR17 subdatasets

We choose the best achieved accuracy for every dataset from all achieved accuracies using different learning rates. The training dataset contains 1182 images. In every of the 394 classes there are 3 specimens. For training on a normal dataset with learning rate 0.01, we achieved accuracy 0.56 after the 38th epoch and for training on a cropped dataset with learning rate 0.1, we achieved accuracy 0.56 after the 56th epoch.

The testing dataset contains 3600 images. In every of the 720 classes there are 5 specimens. For training on a normal dataset with learning rate 0.01, we achieved accuracy 0.39 after the 26th epoch.

The dataset both contains images from the training and the testing ICDAR17 datasets. For training on both datasets together with learning rate 0.0001, we achieved accuracy 0.227 after the 36th epoch.

4.3.2 ICDAR17 results

At first, we will discus the pre-training results. The dataset ICDAR17 without our cropping method has the batch size of 30 samples and for training it has the patch size 100 and for testing it has the patch size 30. The dataset ICDAR17 with our cropping method has the batch size of 30 samples and for

training it has the patch size 3 and for testing it has the patch size 30. We are learning with learning rate 0.1, 0.01 and 0.0001 and the concrete accuracies can be read in table 4.2. At first glance, the concrete numbers can appear to be really bad but we need to realize that we have a lot of classes and only few training images.

Afterwards, we decided to use pre-trained weights from the experiments using the training dataset. It is a smaller dataset, as mentioned before in section 4.3.1, but the accuracy is much better in comparison to the other experiments. Then, we also try to pre-train the model on cropped images from ICDAR17 in the way as was described in the second paragraph of section 4.3. We achieved better accuracy of more than 29 percent. The improvement can be caused by two things, firstly the cropped images are in most cases full of text and secondly we used smaller patches to pre-train the network faster. Normally, there are 100 images randomly cut out of the source image, but we randomly cut out only 3 images because cropping 100 images of size 150×150 from image 224×224 does not seem to be beneficial. After applying the cutting method from part 4.3, we get up to eight images from every image, sometimes less, when the input image is too small.

Model	Pre-trained on	Validation accuracy
ResNet50	ICDAR17 training cropped	0.851
ResNet50	ICDAR17 training	0.560
ResNet50	ICDAR17 testing	0.390
ResNet50	ICDAR17 both	0.227

Table 4.2: Accuracy for pre-training on ICDAR17 dataset

After that we started to fine-tune. We tried to change the learning rate and fix weights of layers from the beginning of the network. However, we did not achieved better results than without pre-training. Usually a small learning rate with the value circa 10^{-5} is used for fine-tuning but when we use this value we get worse results than with a higher value of learning rate. This can be explained by the fact that the ICDAR17 data are too different from the CLaMM16 data.

Model	Pre-trained on	Learning rate	Fixed layers	V. accuracy	Test accuracy
ResNet50	ICDAR17 testing	0.1	0	0.880	0.738
ResNet50	**ICDAR17 training**	**0.1**	**0**	**0.910**	**0.786**
ResNet50	ICDAR17 both	0.01	0	0.920	0.776
ResNet50	ICDAR17 training	0.01	0	0.890	0.774
ResNet50	ICDAR17 training	0.0001	0	0.700	0.597
ResNet50	ICDAR17 training	0.01	25	0.860	0.729
ResNet50	ICDAR17 training	0.01	10	0.905	0.772
ResNet50	ICDAR17 training cropped	0.1	10	0.915	0.761

Table 4.3: Accuracy for fine-tuning on ICDAR17 dataset

When we fixed some layers the learning takes much less time than before. That is one of the pre-training advantages. But in our case the disadvantage is that we get smaller accuracy as it can be seen in table 4.3 above.

4.4 Conclusion

In our case, pre-training is not really beneficial because the dataset was already augmented by using random perturbations affecting the zoom level, rotation angle, shear range, translation in both dimensions and eventually yielding. Also the random cropping helps to generalize and augment the dataset.

During the pre-training, we try to apply another way of pre-processing the data. We identified the main issue of the ICDAR17 dataset and we improved the validation error on the ICDAR17 training dataset from accuracy 0.56 to 0.851 by applying a new cropping method. In this section we got the best result using ResNet50 with learning rate 0.01 without pre-training, concretely the achieved test accuracy was 0.799.

Chapter 5

Other methods

In this chapter, we investigate other methods to improve recognition performance of tasks with scarce data. We will speak about two possibilities of adapting the dataset to achieve better accuracy. The first possibility will be an augmentation technique and the second will be a data preprocessing technique.

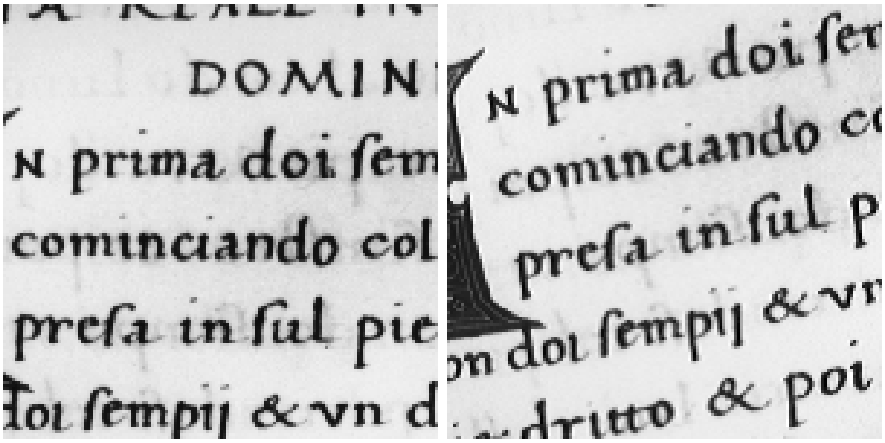
Augmentation techniques changing zoom level, rotation, shear range and translation in both dimensions have been implemented by Mike Kestemont. We chose rotation to describe in more detail. Changing zoom level was in the range $[0.75, 1.25]$, shear range had values in the interval $[-15°, 15°]$ and translation was in the range $[-12, 12]$ pixels. Work on a similar basis is explained in [21]. All of the augmentation techniques can be implemented through a transformation and applied at the same time. Moreover, we added the zero-phase components analysis (ZCA) whitening, which is presented in more detail in section 5.2.

5.1 Rotation

The random generated angles were in the range $(-10°, 10°)$. The rotation can be expressed in a transformation matrix $\boldsymbol{R}$:

$$\boldsymbol{R} = \begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{5.1}$$

Where θ determines rotation angle in axis z. After the rotation, we cropped the image into size 150×150. On the left side, we can see cropped source image 5.1a and on the right side we can see rotated cropped image 5.1b.



(a) : Source image **(b) :** Rotated image by 10°

Figure 5.1: Dataset augmentation using rotation

5.2 Whitening

Whitening decorrelates the image, i.e. redundant pixel information is reduced. There are two basic whitening [22] methods, the first one is principal component analysis (PCA) whitening and the second one is ZCA whitening. We decided to use ZCA whitening method. The calculation of ZCA whitening can be algebraically described by equation 5.2.

$$\boldsymbol{W_{ZCA}} = \boldsymbol{\Sigma}^{-1/2} \tag{5.2}$$

Where $\boldsymbol{W_{ZCA}}$ determines the whitening matrix and $\boldsymbol{\Sigma}$ represents the covariance matrix of one image from the dataset. An important parameter for whitening is small flowting point constant (EPS), which prevents dividing by zero. But when we use a too big EPS then we loose the details of the images and the content can be lost completely.

In the figure 5.2 we can see a concrete example of a whitened image from the CLaMM16 dataset.

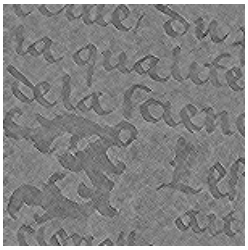


Figure 5.2: Whitened image from CLaMM16 dataset

Moreover, we will describe the concrete way of computing the whitening effectively. At first, we calculate the covariance matrix from the cut-out. Then, we compute eigenvectors ($\boldsymbol{U}$) and eigenvalues ($\boldsymbol{\lambda}$). Afterwards, we write the inverted value of square root of eigenvalues on diagonal of a matrix. We can label this matrix $\boldsymbol{D}$. To prevent dividing by zero, it is needed to add to eigenvalues a small constant, in our case it is 10^{-8}. To get the whitened image, we need to multiple matrix $\boldsymbol{D}$ by eigenvectors from both sides. We can express the calculation as is written in equation 5.3, where $\boldsymbol{EPS}$ represents the vector of EPS. Application of whitening matrix on the source image (A) is mentioned in equation 5.4, where $\boldsymbol{B}$ determines the whitened image.

$$\boldsymbol{W_{ZCA}} = \boldsymbol{U} \cdot \operatorname{diag}\left(\frac{1}{\sqrt{\boldsymbol{\lambda} + \boldsymbol{EPS}}}\right) \cdot \boldsymbol{U}^{\mathrm{T}} \tag{5.3}$$

$$\boldsymbol{B} = \boldsymbol{W_{ZCA}} \cdot \boldsymbol{A} \tag{5.4}$$

When we look at validation accuracy graph with whitening, figure 5.3. It looks similar to the graph without whitening (4.4), but we achieved slightly better results in this case.

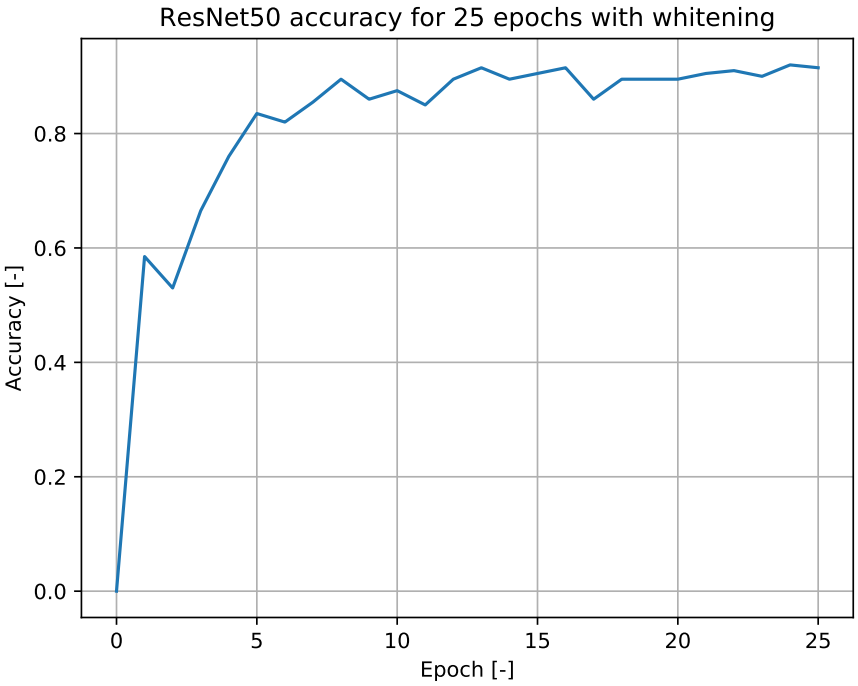


Figure 5.3: Validation accuracy of ResNet50 using whitening

The comparison of other best developed accuracies with the results of whitening method is available in table 5.1. Notice the testing accuracy, which is better for ResNet50 with whitening by 6.4% in comparison with ResNet50 without whitening.

Model	Validation accuracy	Testing accuracy
ResNet50 with whitening	0.920	0.863
ResNet50 without whitening	0.905	0.799

Table 5.1: Comparison of the accuracy with and without whitening

At first, we did not expect from this method much better results than from training without whitening because a kind of normalization is already applied in batchnormalization layers, however, it is applied to whole batches there. In contrast to that, whitening brings normalization to each single image and lets excel the specifics of the image. For training on whitened CLaMM16 dataset with learning rate 0.1 we got after 23rd epoch validation accuracy 0.92 and test accuracy 0.863.

Chapter 6

Conclusion

At first, we have adjusted the CNN system for script style classification by Mike Kestemont [6] for the Friedrich-Alexander Universität (FAU) cluster and current libraries. Afterwards, we adapted the script to use pre-training. We pre-trained the script on the ICDAR17 dataset, however, we got worse results than without pre-training, see tables 4.1 and 4.3.

We also investigated the method of fixing layers of the beginning of the network, but we got even worse results. We can explain this deterioration by the dissimilarity of the pre-training and the fine-tuning datasets. Both of them date to different centuries. Translations and rotations to augment the dataset by Mike Kestemont can help to broaden the dataset enough to train ResNet50, which means that the pre-training was useless and even confused the CNN. We have also tried to change the learning rate. Others have typically got the best testing accuracy for a small learning dataset for fine-tune for the learning rate in order of 10^{-5}. However, we have got the best accuracies for fine-tuning with learning rate 0.1, what we can classify like completely new training with atypical initialization.

We got low validation accuracy for pre-training that is why we used the new method to **crop** the subimages from the whole image (section 4.3) to improve **the validation accuracy** of pre-training the ICDAR17 training dataset **from 0.56 to 0.851**.

During investigation of other methods we got the best results when we **whitened** every single image crop (section 5.2), then **the test accuracy** for the CLaMM16 dataset achieved a value of **0.863**. That is by 9.81%

better than Mike Kestemont's results from the ICFHR2016 competition [3] and in comparison with the best accuracy in the ICFHR2016 competition, DeepScript-whitening better by 2.4%. For all results of competition and our best result please refer to table 6.1.

Team	Testing accuracy	Method
DeepScript-whitening	**0.8630**	**CNN ResNet50 whitening**
FAU	0.8390	i-vector + SVMs
NNML	0.8380	CNN
FRDC-OCR	0.7980	CNN
DeepScript	0.7649	CNN VGG-architecture
TAU-3	0.5280	k-NN out of the training histograms
TAU-2	0.5010	k-NN out of the training histograms
TAU-1	0.4990	k-NN out of the training histograms

Table 6.1: Comparison of results of Bachelor Thesis and of the ICFHR2016 competition

In the future, whitening in the CLaMM16 dataset with the calculation of one covariance matrix over all the pictures from the training dataset together can be investigated or layer-specific adaptive learning rates can be also tried. We can also classify other aspects of Latin Handwritings, e.g. the dating of historical documents.

If you want to verify the results yourself, if you want your CNN to learn, then follow the steps in appendix B.

Bibliography

[1] SIMONYAN, Karen; ZISSERMAN, Andrew. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556, 2014.*

[2] HE, Kaiming, Xiangyu ZHANG, Shaoqing REN and Jian SUN. Deep Residual Learning for Image Recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR).* IEEE, 2016, , 770-778. DOI: 10.1109/CVPR.2016.90. ISBN 978-1-4673-8851-1.

[3] CLOPPET, Florence, Véronique EGLIN, Van KIEU, Dominique STUTZMANN and Nicole VINCENT. *ICFHR2016 Competition on the Classification of Medieval Handwritings in Latin Script.* Shenzhen, China, 2016, 2016(Oct.).

[4] GOODFELLOW, Ian, Yoshua BENGIO and Aaron COURVILLE. *Deep Learning.* Massachusetts: MIT Press, 2016.

[5] STUTZMANN, Dominique. *ICFHR2016 competition on the classification of medieval handwritings in latin script* [online]. 2016 [cit. 2017-07-07]. Available from: https://oriflamms.hypotheses.org/1388

[6] KESTEMONT, Mike *DeepScript*[online]. Brusels, 2016 [cit. 2017-06-14]. Available from: https://github.com/mikekestemont/DeepScript

[7] CHOLLET, François. *Keras: The Python Deep Learning library* [online]. [cit. 2017-07-07]. Available from: https://keras.io/

[8] *TensorFlow: An open-source software library for Machine Intelligence* [online]. [cit. 2017-07-07]. Dostupné z: https://www.tensorflow.org/

[9] RUDER, Sebastian. AYLIEN. *Transfer Learning: Machine Learning's Next Frontier* [online]. 2017 [cit. 2017-07-04]. Available from: http://sebastianruder.com/transfer-learning/

[10] DEHAK, Najim, Pedro A.TORRES-CARRASQUILLO, Douglas REYNOLDS and Reda DEHAK. *Language Recognition via Ivectors and Dimensionality Reduction.* INTERSPEECH. 2011.

[11] KARPATHY, Andrej. STANFORD UNIVERSITY. *CS231n: Convolutional Neural Networks for Visual Recognition* [online]. [cit. 2017-07-03]. Available from: http://cs231n.github.io/

[12] DESHPANDE, Adit. *A Beginner's Guide To Understanding Convolutional Neural Networks Part 2* [online]. 2016 [cit. 2017-07-07]. Available from: https://adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks-Part-2/

[13] TAPIA, Guido. *Review of Keras (Deep Learning) Core Layers* [online]. 2016 [cit. 2017-07-07]. Available from: http://www.picnet.com.au/blogs/guido/post/2016/05/16/review-of-keras-deep-learning-core-layers/

[14] IOFFE, Sergey and Christian SZEGEDY. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift.* 2014, 3-5.

[15] YANG, Ji. *ReLU and Softmax Activation Functions* [online]. 2017 [cit. 2017-07-07]. Available from: https://github.com/Kulbear/deep-learning-nano-foundation/wiki/ReLU-and-Softmax-Activation-Functions

[16] BISHOP, Christopher M. *Pattern recognition and machine learning.* New York: Springer, c2006. Information science and statistics. ISBN 03-873-1073-8.

[17] RUDER, Sebastian. *An overview of gradient descent optimization algorithms.* 2017, 1-5.

[18] KARPATHY, Andrej. Learning rates. In: *CS231n: Convolutional Neural Networks for Visual Recognition* [online]. Stanford university [cit. 2017-07-06]. Available from: http://cs231n.github.io/assets/nn3/learningrates.jpeg

[19] KARPATHY, Andrej. Accuracies. In: *CS231n: Convolutional Neural Networks for Visual Recognition* [online]. Stanford university [cit. 2017-07-06]. Available from: http://cs231n.github.io/assets/nn3/accuracies.jpeg

[20] CHRISTLEIN, Vincent, Martin GROPP, Andreas MAIER and Stefan FIEL. *Unsupervised Feature Learning for Writer Identification and Writer Retrieval.* 2017.

[21] KLEP, Denise. *Data augmentation of a handwritten character dataset for a Convolutional Neural Network and integration into a Bayesian Linear Framework.* Nijmegen, 2016. Bachelor Thesis. Radboud University. Supevised by Sanne Schoenmakers Marcel van Gerven.

[22] KESSY, Agnan, Alex LEWIN and Korbinian STRIMMER. *Optimal Whitening and Decorrelation. The American Statistician.* 2017, 2017(1), 0-0. DOI: 10.1080/00031305.2016.1277159. ISSN 0003-1305.

Appendix A

Acronyms

BGD Batch gradient descent. 17, 18

CLaMM16 dataset of handwritings. vii, viii, 20, 26, 27, 29–32, 36–40, 48

CNN convolutional neural networks. iv, vii, 1–3, 5–8, 10, 16, 19, 23, 24, 26, 30, 39, 40, 47

CONV convolutional. 26

CPU central processing unit. 6, 19

EPS small flowting point constant. 36, 37

FAU Friedrich-Alexander Universität. 39

FC fully connected. 26

FEL Faculty of Electrical Engineering. iii

GPU graphics processing unit. 6, 19

ICDAR17 dataset for competition on the classification of medieval handwritings in Latin script. viii, 26, 29, 31–33, 39

kNN k nearest neighbors. 20

MBGD mini-batch gradient descent. 17, 18

NAG Nesterov accelerated gradient. 18

NN neural network. 1, 5, 7, 13, 20

PCA principal component analysis. 36

ReLU Rectified linear unit. vii, 8, 13–15

RGB color model. 30

SGD stochastic gradient descent. 4, 17, 18

SVM support vector machine. 3, 20

tanh hyberbolic tangent. 13

WCCN within class covariance normalization. 3

ZCA zero-phase components analysis. 35, 36

Appendix B

Instructions for running the scripts

The files on CD were developed with Python programming language, version 3.5.2, with additional libraries Keras, version 1.2.2., that run on Tensorflow backend. Use of Keras in version 1.2.2 is obligatory, there is currently a bug in later version that brings many difficulties with dimensions ordering in image during the learning. The last version does not react to settings of **image_dim_ordering** in **keras.json** file.
Secondly it is needed to have the right configuration written in the configuration file for Keras. The file called **keras.json** must look as shown in following listing.

```
{
        "image\_dim\_ordering": "th",
        "epsilon": 1e-07,
        "floatx": "float32",
        "backend": "tensorflow"
}
```

You can find file **keras.json** in hidden directory **home/.keras/**. For learning weights of a CNN, it is recommended to run the file **trainGeneral.py** with three arguments: name of the directory in directory model, type of architecture (for VGG16 write vgg16 and for ResNet50 write resnet50) and path to learning data. For example to train by uncropped testing part of the ICDAR17 dataset use the command shown in following listing.

```
python3 trainGeneral.py Resnet50-ICDAR17-Normal-test
    ↪ resnet50 /data/all/ICDAR17/normal/test
```

Files **weights.hdf5**, with learned weights, **architecture.json**, where we can clearly see used architecture of our network (in our example resnet50), and **labelencoder.p**, which is needed for library sklearn, because LabelEncoder() only takes a 1-d array as an argument can be found in directory **model/Resnet50-ICDAR17-Normal-test**.

If you want to fine-tune the dataset use also the script trainGeneral.py but with five arguments. First three arguments have the same meaning as for training. The forth argument is True, if we want to use fine-tuning and False, if we do not want to fine-tune. The fifth argument says in which directory of the directory model we find the pre-train architecture and pre-trained weights. The example in following listing fine-tunes the pre-trained model from the directory resnet-ICDAR17-normal-train on dataset /data/splits/ for type model resnet50 and saved weights, architecture and label encoder in directory resnet-ICDAR17-normal-train-finaltuneLR0-0001.

```
python3 trainGeneral.py resnet-ICDAR17-normal-train-
    ↪ finaltuneLR0-0001 resnet50 /data/all/CLaMM16/
    ↪ True resnet-ICDAR17-normal-train
```

In order to test run the file **testGeneral.py** with 2 arguments, first is model (same as directory in directory model in attached source code) that you want to test and second is the dataset on which you want to test. For example, to test the model final on CLaMM16 dataset run command shown in following listing.

```
python3 testGeneral.py final /data/all/CLaMM16/test
```

If we want to run testing and training for whitened data, we need to run both scripts trainGeneral.py and testGeneral.py in special way. In both cases we add an argument True, in case of training as 4th argument and in case of testing as 3rd argument. Meanings of the rest of the arguments are explained above. We can read an example for whitening in following two listings.

```
python3 trainGeneral.py resnet-patch-Whitening resnet50
    ↪ /data/all/CLaMM16/ True
```

```
python3 testGeneral.py resnet-patch-Whitening /data/all
    ↪ /CLaMM16/test True
```

Final note: all paths are written relatively to the location of the script that you run.

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Cybernetics

BACHELOR PROJECT ASSIGNMENT

Student: Zuzana K o ž n a r o v á

Study programme: Cybernetics and Robotics

Specialisation: Robotics

Title of Bachelor Project: Manuscripts Classification Using Convolutional Deep Learning Networks

Guidelines:

In recent years, deep learning techniques based on neural networks (NN) have set new performance standards for a wide variety of machine learning or computer vision tasks. However, one problem of these deep NNs are there need of many training samples. Thus, for small or medium-sized datasets, classical computer vision approaches still surpass NNs. For example for the task of medieval script type classification, methods based on convolutional neural networks (CNN) do not outperform classical approaches based on 'bag of visual words'. The goal of this work is to improve the performance of CNNs for script type classification. A possible solution to achieve better results is to pretrain CNNs for a different, yet similar task, and fine-tune it later with the training data of the original task. Possible related tasks could be the dating of historical documents, or writer identification. For both related tasks, datasets will be provided. In detail, the thesis consists of the following parts:

- the implementation of a CNN system for script style classification (it can be based on existing work, e.g.),
- evaluate the usefulness of pre-training for script-style classification using other datasets,
- investigate other methods to improve recognition performance of tasks with scarce data.

The implementation should be in Python using the deep learning frameworks Keras (and / or Tensorflow).

Bibliography/Sources:

[1] Goodfellow, Ian et al.: "Deep Learning", MIT Press 2016, online available http://www.deeplearningbook.org
[2] Szeliski, Richard: "Computer Vision. Algorithms and Applications", Springer, 2011, ISBN: 978-1-84882-934-3, online available at http://szeliski.org/Book/
[3] Duda, Richard O.; Hart, Peter E.; Stork, David G.: "Pattern classification". John Wiley & Sons, 2012.
[4] Bishop, Christopher M.: "Pattern recognition." Machine Learning 128 (2006).

Bachelor Project Supervisor: Dipl.-Inf. Vincent Christlein

Valid until: the end of the summer semester of academic year 2017/2018

L.S.

prof. Dr. Ing. Jan Kybic
Head of Department

prof. Ing. Pavel Ripka, CSc.
Dean

Prague, January 14, 2017